


Overcoming Obstacles to Mainlining

Tim Bird
Senior Software Engineer
Sony Mobile Communications

© Sony Mobile Communications

Agenda

1. Identify obstacles to mainlining
 2. ???
 3. Profit!
- Overcome Obstacles
- 

Agenda

- Identifying obstacles
 - Survey
- Describing obstacles
 - Observed
- Overcoming obstacles
- Best Practices
- Incentives (Profit!)

Identifying Obstacles

Identifying Obstacles

- A side track on philosophy...
- Survey
 - Some quantifiable data (on perceived issues)
- Observed obstacles

Tolstoy and Bera

- Anna Karenina Principle
 - "Happy families are all alike; every unhappy family is unhappy in its own way"
 - There are lots of ways to fail, but only a few ways to succeed
- Yogi Bera (American baseball player, philosopher)
 - "If people don't want to come out to the ballpark, nobody's going to stop them."
 - Motivation is a key element

Survey

- Conducted an online survey in September 2014
- Goal was to find qualified kernel developers, who do NOT submit patches upstream
 - And determine “why not?”

Survey results

- Top obstacles:

Obstacle	General rating	Industry rating
Older kernel version	48%	56%
It's too hard	47%	24%
Patch not good enough	44%	27%
Employer does not provide time	40%	34%
Afraid of rejection	35%	15%
Depends on other code not upstream	35%	46%
Could not test	33%	42%

Some more insights from survey

- Developer motivation:
 - It is important to submit change upstream: 92%
 - I would like to submit changes upstream: 91%
- Management motivation:
 - Management doesn't approve: 21%
 - Employer doesn't provide time: 40%

Interesting non-issues:

- English not good enough: 9%
- Not my responsibility: 6%
- Company process too hard: 26%

Some interesting quotes

- Referring to the company approval process: “It can take weeks or months to get a commit out for contribution”
- “[We] mainly work on older kernels with our supplier’s modifications”
- “It is not really clear what direction a newbie should follow after... the first few patches...”
- “Drop the hard words/language on LKML...”

Obstacles summary

- Version gap (working on older kernel)
- Perceived difficulty
- Low-quality or specialized code
- Dependency on non-mainlined code
- Not enough time

Describing Obstacles

Version gap

- Many companies use a vendor tree
 - Particularly true for products with Android
- Are locked in because of processor or SOC selection
- Some amount of patches on top of vanilla
- Development/Testing/Release schedules causes delay in kernel version

Example of version gap

- Delta between Sony Mobile and mainline
 - Sony mobile dependent on upstream supplier for Linux version (3.4 in this case)
 - Lots of patches between Sony tree and mainline

Committer e-mail	Commits	Authors
Google/Android commits	963	61
Other	2677	828
Qualcomm	20395	635
Sony Mobile	1799	203
Between our tree and mainline base (3.4)	25843	1757

- Haven't determined at which stage "Other" contributions are integrated.
- Haven't quantified how many Sony Mobile patches are dependent on non-mainline
 - Rough estimate is that most (~90%) are

Perceived difficulty

- Process is cumbersome if you are not familiar
- List of requirements for a contribution is long
 - SubmittingPatches, SubmitChecklist, CodingStyle
 - Good, but don't cover a variety of social issues
 - Getting anything wrong can result in failure
 - Lots of details which maintainers take for granted
- Not as strict as it used to be, and there are now tools to assist (e.g. checkpatch.pl)
- Cause of strictness is maintainer overload – don't have time for malformed contributions
 - Silly mistakes is the first filter

Perceived difficulty (cont.)

- Part-time contributions
 - Switching cost of juggling between contributing and product development is high
 - Similar to high-latency scheduling – results in overall poor performance
 - Not doing full-time contributing means that proficiency in open source methods is developed slowly
 - Can result in bad response time to provided feedback
- Classic error:
 - Working on a large patch in isolation
 - Attempt to mainline and find that major changes are needed
 - Results in mantra: “release early and often”
 - Original development strategy made it hard

Low-quality or specialized code

- Low-quality
 - Workarounds and quick hacks
- Specialized code
 - Not generalized for other use cases
- Sometimes, there are no frameworks, or the framework is immature
 - E.g. NFC support for Android
- Assumption by developer (probably correct) that refactoring of submitted code or even refactoring of upstream code is required to accept the change in mainline

Dependency on non-mainlined code

- Modifications to drivers and systems that are not upstream
 - Bugfixes and workarounds for code not upstream
 - It's unclear where to send fixes
 - If it's an IP block in an SOC, who should get the fixes?
 - SOC vendor?, IP block creator?
- Example: bugfixes for synaptics touchscreen driver
 - Long delays getting synaptics driver upstream
 - Impractical, and low motivation to do mainlining in place of hardware supplier

Not enough time

- Not enough time provided by management
- Product teams focused on tight delivery deadlines
- Causes focus on “good enough” solutions
 - Not unique to open source software
- No time to respond to change requests
- I refer to this as the “product treadmill”
- Mainline versions are independent of any notion of product release dates
 - Mainline acceptance happens when it happens, not based on your need

Observed obstacles

- Required expertise is very high (and increasing)
 - This is true for core systems, but not drivers
 - Proxy problem – someone other than author is contributing the code (will be discussed later)
- Internal Linux churn
 - Linux has no ABI or even stable API internally
 - This is a root cause of version gap issues
- Specialized code (often hacks)
 - Code for just one hardware or one product release
 - Attitude that code is “throwaway”, or that code is “good enough” for one embedded product release
 - Assumption that reuse is not needed

Overcoming Obstacles

Overcoming Obstacles

- Solution for version gap:
 - Get a minimal core of mainline running on your hardware
 - Have one team working on mainline, while product engineers work on older kernel (creates the proxy problem, described later), until you catch up
- Solution for product treadmill
 - Small team dedicated to mainline, off of product treadmill
- Solution for perceived difficulty
 - Internal training, mentors
 - Use same processes internally as upstream
 - Avoid re-learning upstream methods

Overcoming Obstacles (cont.)

- Solution for low-quality code
 - Quick hacks are sometimes appropriate from a cost/benefit standpoint
 - Need to determine whether code should be upstreamed
 - Measure duration in your internal tree, and re-work hack if you are carrying it from release to release
 - Maybe tag such hacks so they can be tracked?
- Solution for specialized code
 - Do better at sourcing
 - Require mainline Linux drivers from hardware supplier
 - Actually consider software cost in BOM (I can dream can't I?)
 - Only industry working together can work on this

The Proxy Problem

- Open-source-facing developers are experienced with the hardware or system that needs to be mainlined
- Is when your “proxy” tries to mainline something
 - Doesn't have in-depth knowledge of change
 - Can't answer questions in a timely manner
 - May not be able to test thoroughly
- In case where your change is too far from mainline, you may just have a code impedance problem
 - Details matter (e.g. locking)
- Some possible solutions
 - Proxies mentor original developers to have them mainline the code
 - Original developers assist proxies in understanding and testing

Best Practices

From the kernel gurus

Andrew Morton tips

- See Andrew Morton's ELC 2008 Keynote:
 - http://elinux.org/Session:kernel.org_development_and_the_embedded_world
- Industry should have an embedded maintainer
- Report problems and requirements upstream
- Participate in community forums
- Companies should dedicate a few developers separate from product teams
- Develop product on latest mainline kernel, freeze it at end of product development
 - My aside: Current nature of Android features and board support preclude this
- Ask the community (Andrew) for help

Deepak Saxena tips

- Don't be arrogant
 - Don't assume you know better than community developers
- Release early and often
 - Don't work in isolation, and then make big changes when submitting
- Do your homework
 - Check for existing solutions and extend those
- Don't add OS abstractions (or, HALS for other OSes)
- Write general solutions
- Learn community methods
- Work with the community
 - Treat them as equals on your team

Jonathan Corbet tips

- Post early and often
- Submitting patches
 - Send changes - can influence direction even if not accepted
 - No: multi-purpose patches - make each patch small and independent
 - Make patch serieses bisectable
 - Follow submission and style rules
 - Send to correct place: MAINTAINERS, get-maintainer.pl
 - Listen to reviewers, be polite, don't ignore feedback
- Be open to accepting changes
 - Your code may be re-written or replaced
- Coding
 - Follow the style guidelines
 - No multi-OS code – no HAL layers, unused parameters
 - Should generalize existing code instead of create new code, where possible
 - Don't break APIs to user space
 - Don't cause regressions

Incentives

Incentives

- Why study this?
 - Sony Mobile has 1100 people who have made a patch to the kernel
 - We find ourselves applying the same changes over and over again
- Would like to decrease number of kernel developers by moving stuff to mainline
 - OR – have them move to different tasks (power enhancement, performance, etc.)

Profit!

Committer e-mail	Commits	Authors
Google/Android commits	963	61
Other	2677	828
Qualcomm	20395	635
Sony Mobile	1799	203
Between our tree and mainline base (3.4)	25843	1757

- Reduce maintenance cost
 - Allows others to maintain and enhance the code
- Reduce time to market
 - Even more important than cost

Reasons to submit upstream

- Improves code quality
 - You get immediate feedback, even if code is not accepted
 - It gets more long-term testing
- Avoids adopting a competing implementation
 - Have 3rd parties enhance your implementation rather than something else
- It rewards your developers
 - They want to contribute, for a variety of reasons
 - They become better developers through interaction with the community
- Please notice these are selfish reasons
 - Unselfish reasons are valid also

Factors for overcoming (from survey)

Factor	General Rating	Industry Rating
Better maintainer feedback	48%	21%
Time dedicated by employer	44%	74% !!
Instruction or training	39%	32%
Mentoring	37%	32%
Permission from employer	25%	52%

Key recommendations:

- Work with SOC supplier to reduce version gap
- Have a dedicated team that works in open source
- Do specific training for:
 - Better motivation (management training)
 - Open source methodology and tactics
- Consciously work on social element of community engagement
 - Work on stuff for others, and they'll help you
 - Meet maintainers face-to-face if possible
 - Conferences are helpful for this

CE Workgroup Device Mainlining Project

- BOF – Thursday 4:00 pm

Resources

- http://elinux.org/Kernel_Mainlining